Heinrich Heine
Universität
Düsseldorf

INSTITUT FÜR INFORMATIK
Softwaretechnik und Programmiersprachen
Universitätsstr. 1 D–40225 Düsseldorf

A general-purpose modifiable AST parser

Michael Doberstein

Bachelorarbeit

Beginn der Arbeit:	29th of September 2024
Abgabe der Arbeit:	30th of December 2024
Gutachter:	Dr. John Witulski
	Dr. Jens Bendisposto

Ehrenwörtliche Erklärung

Hiermit versichere ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, den 30th of December 2024

Michael Doberstein

Abstract

Software developers are often facing the issue of maintaining large amount of source code. The keep the source code up to date, often times a process called refactoring is needed, in which the software developers update the code.

To allow the software developers to perform these updates automatically, most integrated development environments (IDE) provide tools to perform predefined refactorings automatically. The effort of performing some refactorings on large code bases can be too much to perform manually. Most of the times, the tools provided by the IDE are sufficient to perform the refactorings, but sometimes a custom solution is required. One way to create automated custom refactorings is by parsing the source code and building it into an abstract syntax tree (AST), which can be maniup

The tools given by an IDE or other sources are sometimes not sufficient to perform an automated refactoring, therefore other methods of generating custom refactorings are required.

Here we provide a Java library, which can parse a subset of the context-free language based on their corresponding grammar definition and generates a AST which can be modified by the user and afterwards reverted back to the source code.

The program provides an relatively easy method of safe large scale refactorings, however the effort of defining a grammar is large. It also has been shown, that this tool can also be used to extend a given programming language and allow for additional features to be added, basicly allowing for an primitive form of source to source compilation. We can also see clear limits with this tool, as it does not include features that are specific to each programming language, like symbol tables, which are not defined by the grammar of the language. This limits the amount of refactorings that can be archived without implementing additional functionality on top of the base program.

Erklärung zur Nutzung generativer KI

Im Rahmen der vorliegenden Bachelorarbeit wurde generative KI zu folgenden Zwecken genutzt:

- DeepL Write zur sprachlichen Überarbeitung der Arbeit
- Machine learning assistant von der IntelliJ IDE wurde bei Implementierung des Programmes verwendet

Acknowledgements

Im Falle, dass Sie Ihrer Arbeit eine Danksagung für Ihre Unterstützer (Familie, Freunde, Betreuer) hinzufügen möchten, können Sie diese hier platzieren.

Dieser Part ist optional und kann im Quelltext auskommentiert werden.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Related work 1

2 Prerequisites **1**

2.1 Regular Expressions and Grammars 1

2.2 Regular Expressions and Grammars 2

2.3 Parser 4

2.3.1 LR(1)-Parsing 4

3 Implementation **5**

3.1 Architecture 5

3.2 Class generation vs general classes 6

3.3 Grammar file syntax 7

3.4 CST-Generation 8

3.5 AST-Generation 9

3.5.1 list-Modifier 9

3.5.2 alias-Modifier 10

3.5.3 hidden-Modifier 11

3.5.4 inline-Modifier 11

3.5.5 Not implemented modifiers 11

3.6 AST search methods 12

3.7 Selectors 12

3.8 AST Modifications 14

3.9 Extendability 14

3.9.1 Lexer 14

3.9.2 AST Generation 15
3.9.3 Selectors 16

4 Grammars **17**
4.1 MiniJava 17
4.2 Extended MiniJava Subset 17
4.3 Automated Refactoring 17

5 Use cases **17**
5.1 Yoda conditions 18
5.2 Unreachable code removal 18
5.3 Translation system 18
5.4 Code transpilation 19
5.5 Code analysis 20

6 Conclusion **20**

List of Figures **21**

List of Tables **21**

References **22**

1 Introduction

1.1 Motivation

1.2 Related work

There are many parser generators that have a similar approach, but different goals. One major example is ANTLR, which generates a parser for a given grammar. The generated parsers are mainly used for further use in a compiler. This usecase differs from the usecase of refactoring, as the compiler does not need whitespaces, comments or preprocessor statements in most cases. In the use case of refactoring, we need to preserve all of these tokens to reconstruct the source code as accurately as possible. We do not want to loose all comments due to a refactoring beeing made. Also, this thesis provides a tool which provides an API specifically aimed to refactor source code.

2 Prerequisites

In this chapter we want to establish the prerequisites required to understanding this thesis.

2.1 Regular Expressions and Grammars

In theoretical computer science, a grammar consists out of a set Σ of terminal symbols, a set N of nonterminal symbols, a set P of productions and a start symbol S. S is always a nonterminal symbol and therefore $S \in N$. The productions are defined as $P \subseteq (N \cup \Sigma)^+ \times (N \cup \Sigma)^*$. The $^+$ refers to one or more, the * refers to zero or more.

Grammars can be seperated in different different classes based on the chromsky hierarchy. The main ones we are looking at in this thesis are the contextfree grammars and the regular grammars.

A grammar is context free, if for all productions $p \to q$ in P the following applies: $p \in N$.

A grammar is a regular grammar, if for all prodution $p \to q$ in p the following applies: $p \in N$ and $q \in \Sigma \cup \Sigma N$.

Grammars are used, because they can describe a language. In our use case, they describe an programming language and can be used, to parse the source code. Regular expressions are short statements, that avoid defining an entire grammar definition. Their

downside is, that they are not able to describe alot of the features modern programming languages have. For example, it is not possible to write a regular grammar for the language that has the same amount of opening and closing brackets. An example for a regular expression would be this:

$$a^*ba^*b$$

This would parse all words that have any amount of "a", followed by exactly one b, followed by any amount of "a", followed by exactly one b.

To handle these situations, we require the context free grammars, they are able to parse more cases and are sufficient enough, to parse most programming languages. However, even the context free grammars are not powerfull enough to correctly verify if a given source code is valid. For this, we would technically need context sensitive grammars. Those come with the cost of a worse runtime, therefore usually context free grammars are used when writing compilers and additional verifications are done after the parsing to ensure the language is indeed valid source code.

An example for a context free grammar:

Let $\Sigma = \{E\}, N = \{a, b\}, S = \{E\}$ and the productions defined as:

$$\begin{aligned} P = \{&E \to aEb, \\ &E \to \epsilon\} \end{aligned}$$

ϵ is a special symbol, it refers to the empty word. That means, that E can be derived to nothing. Without this, we would have endless self recursion in the grammar rule.

This grammar would be equivalent to the opening and closing bracket example from previously. This will parse any string, which has for one opening a exactly one opening b.

2.2 Regular Expressions and Grammars

In theoretical computer science, a grammar consists out of a set Σ of terminal symbols, a set N of nonterminal symbols, a set P of productions and a start symbol S. S is always a nonterminal symbol and therefore $S \in N$. The productions are defined as $P \subseteq (N \cup \Sigma)^+ \times (N \cup \Sigma)^*$. The $^+$ refers to one or more, the * refers to zero or more.

Grammars can be seperated in different different classes based on the chromsky hierarchy. The main ones we are looking at in this thesis are the contextfree grammars and the regular grammars.

A grammar is context free, if for all productions $p \to q$ in P the following applies: $p \in N$.

A grammar is a regular grammar, if for all prodution $p \to q$ in p the following applies: $p \in N$ and $q \in \Sigma \cup \Sigma N$.

Grammars are used, because they can describe a language. In our use case, they describe an programming language and can be used, to parse the source code. Regular expressions are short statements, that avoid defining an entire grammar definition. Their downside is, that they are not able to describe alot of the features modern programming languages have. For example, it is not possible to write a regular grammar for the language that has the same amount of opening and closing brackets. An example for a regular expression would be this:

$$a^*ba^*b$$

This would parse all words that have any amount of "a", followed by exactly one b, followed by any amount of "a", followed by exactly one b.

To handle these situations, we require the context free grammars, they are able to parse more cases and are sufficient enough, to parse most programming languages. However, even the context free grammars are not powerfull enough to correctly verify if a given source code is valid. For this, we would technically need context sensitive grammars. Those come with the cost of a worse runtime, therefore usually context free grammars are used when writing compilers and additional verifications are done after the parsing to ensure the language is indeed valid source code.

An example for a context free grammar:

Let $\Sigma = \{E\}, N = \{a, b\}, S = \{E\}$ and the productions defined as:

$$\begin{aligned} P = \{&E \to aEb, \\ &E \to \epsilon\} \end{aligned}$$

ϵ is a special symbol, it refers to the empty word. That means, that E can be derived to nothing. Without this, we would have endless self recursion in the grammar rule.

This grammar would be equivalent to the opening and closing bracket example from previously. This will parse any string, which has for one opening a exactly one opening b.

2.3 Parser

The second part will perform the parsing on the token stream and by using the grammar rules. There are many kinds of parsers, the most common ones used in compiler development are:

1. LL(1): Scanning the input from left to right, applying leftmost deriviation, using one token of lookahead.
2. LR(1): Scanning the input from left to right, applying rightmost deriviation, using one token of lookahead.
3. LALR(1): A variation of LR(1), which reduces the amount of calculations and memory requirements.

There are two goals of the parsing process. The first one is to verify that the input matches the grammar. The second one is to generate a tree structure, which represents the input. The tree structure generated by the parser is called a concrete syntax tree (CST). This tree structure is very verbose and represents the grammar definition exactly.

There are two main different ways used to parse source code. The first approach is top-down parsing. In top-down parsing, the root node will be generated first, following by the childrens and the leaf nodes will be generated at the end. LL(1) parsing is one example for top-down parsing. The other approach is bottom-down parsing. In this parsing strategy the leaf nodes are generated first and the root node is generated at the very end. LR(1) and LALR(1) are examples for bottom-down parsing algorithms.

The top-down parsers have the benefit of being rather easy to understand, however they are less powerfull. Bottom-down parsers are harder to understand and to debug, however they can parse more grammars.

Each of the presented grammars are not able to parse all context free grammars, but they are sufficient enough to parse most programming languages. LR(1) is the parsing strategy that is able to parse the most grammars, but requires the largest amount of memory and calculations to correctly parse.

As memory and runtime is not the main priority in this thesis, a LR(1) parser was implemented to allow the largest amount of grammars being parsed.

2.3.1 LR(1)-Parsing

The LR(1) parsing strategy relies on two different tables. The first table is called an action table. The rows represents different states. The columns represent the terminal symbols

of the grammar. The value of each cell can be empty, a shift, a reduce or an accept action.

A shift action will consume a token from the token stream and move to a new state.

A reduce action will go back to a previously encountered state and references a grammar rule that is beeing reduced.

A accept action is only defined once in the entire table. Once this is encountered, the parsing process is finished and the input was successfully parsed.

The second table is the goto table. The rows represents the different states. The columns represent the non terminal symbols of the grammar. The content of each cell can either be a reference another state or empty.

The parser itself manages a stack of states. The top of the stack is the current state that is currently being processed. The parser reads the current token in the token stream provided by the lexer and receives the current action from the action table. Based on the action type, different behaviours apply.

On a shift action, the parser will shift the position in the token stream and will continue in the next iteration with a new token. The new state defined in the shift action will be pushed on the stack. Then the next iteration is started.

On a reduce action, the parser will pop as n states from the stack, where n is equal to the amount of terminal and nonterminal symbols on the right hand side of the grammar rule. Afterwards, the new top of stack is beeing read and the state defined in the goto table of the top of stack and the current token in the token stream will be pushed on the stack.

On an accept action, the parser will finalize the parsing process accept the input.

3 Implementation

The program is implemented as a Java library and can be found in this github repository: `https://github.com/midob101/gp-modifiable-ast`

3.1 Architecture

The only dependencies used are JUnit and JUnit-jupiter [jun] for testing purposes.

The implementation is split in several packages.

- config_reader: responsible for parsing the language definition files

- grammar: provides classes to store the parsed grammar production rules.
- language_definitions: provides an interface to receive various definitions, like the production and lexer rules of a given language.
- lexer: contains the lexer process
- logger: internal logger class
- parser: contains the LR(1) parser implementation
- selectors: contains basic selectors and interfaces to define custom ones
- syntax_tree: contains classes for the CST, AST and the CST to AST conversion

The typical workflow will be as followed:

1. Parse a language definition, receive the language definition classes
2. Start the lexer process for a source file, receive the token stream
3. Start the parser process with the token stream, receive the CST
4. Start the process to transform the CST to an AST, receive the AST
5. Query the AST with selectors, receive nodes of the AST
6. Modify found nodes
7. Receive text from the AST, overwrite source file with the modified data

3.2 Class generation vs general classes

Most parser generators will create classes for each production. This allows the user to modify the generated classes, for example by adding new methods. This allows better usability if done correctly, and better performance. Instead of generating code for the parser, gp-modifiable-ast will use generic classes that are the same for all grammars being parsed. This approach does not require any additional code generation by the parser and allows an easier introduction to the system, since a language definition and a source file can simply be passed to the parser generator and an AST is generated. Otherwise, the parser must first be generated and then the parser must be called to parse the source file.

3.3 Grammar file syntax

The productions and lexer definitions are defined separately from the implementations in their own text files. Each language has exactly one text file defining the productions and lexer definitions. The contents are divided into four different chapters.

The following example shows a grammar for arithmetic expressions that could be found in a programming language grammar.

This grammar allows to parse simple arithmetic expressions and allows comments and whitespaces.

Listing 1: Grammar file example

```
LANGUAGE_DEF
    grammar_start = S;

LEXER_RULES
    add = "+";
    subtract = "-";
    multiply = "*";
    divide = "/";
    bracket_open = "(";
    bracket_close = ")";
    integer = regex(\d+);
    whitespace = regex(\s+);
    single_line_comment = customMatcher(singleLineCommentMatcher);
    multi_line_comment = customMatcher(multiLineCommentMatcher);

HIDDEN_LEXER_RULES
    whitespace, single_line_comment, multi_line_comment;

PRODUCTIONS
    S   -> S[alias=left] add T[alias=right] |
           S[alias=left] subtract T[alias=right] |
           T;
    T   -> T[alias=left] multiply F[alias=right] |
           T[alias=left] divide F[alias=right] |
           F;
    F   -> bracket_open S bracket_close |
           integer;
```

This grammar file would be able to parse the following sample code:

Listing 2: Grammar file example

```
// Arithmetic grammar test
```

```
(5+10) /** inline comment */
* (15*20)
```

The first chapter is `LANGUAGE_DEF`. This contains some general definitions for the language. This chapter contains e.g. which comment styles are allowed and how they are identified, the name of the language, the file extension used, whether the language is case-sensitive, and which production should be used as the starting production, whether the language is case-sensitive or not, and which production should be used as the starting production. All settings here are defined as key-value pairs.

The second chapter is the `LEXER_RULES`. This defines all the tokens that can occur in the source file and that the lexer should handle. The lexer can handle three different types of definitions. The first type matches a fixed string. The second type matches a regex. The last type uses a custom implementation to check for a match. The last matching definition wins. For example, the token definition `true_literal = "true";` and the definition `identifier = regex([a-zA-Z_][a-zA-Z_0-9]*);` would both match the string `true`. Therefore, the more general `identifier` token should be defined before the `true_literal` token. This will cause the lexer to recognize the text `true` as `true_literal`. A lexer key must be written in all lowercase (and uppercase) letters.

The chapter `HIDDEN_LEXER_RULES` is a comma separated list of lexer definition keys, that should not be handled in the parsing process and should not be visible in the AST by default.

The chapter `GRAMMAR_RULES` contains all productions of this language. The syntax is similar, but not identical, to the backus natus form. (TODO: Source). The syntax has been extended with most of the modifiers suggested by [OJ09]. The terminals are the keys of the lexer definitions. The nonterminals are defined by the left sides of the productions. All nonterminals must be written in uppercase and underscore. `EPSILON` is used for the empty word. Each terminal and nonterminal symbol can be extended by a list of modifiers in square brackets after the name. `S -> S[alias=left] add T[alias=right]` defines the production with `S` on the left and `S add T` on the right. `S` refers to the same production, `T` refers to a different production. `add` refers to a lexer definition. `[alias=right]` sets an alias in the AST for the `T` production.

3.4 CST-Generation

The concrete syntax tree (CST) is a tree structure that exactly matches the grammar and lexer definitions. The generated CST should also contain nodes that are usually omitted, such as comments or whitespace.

The CST is implemented by a `ConcreteSyntaxTreeNode` class. A node can reference either a production or a lexer token. We keep the productions, as they allow us to make advanced modifications and generate a less verbose abstract syntax tree from the CST.

The lexer tokens are maintained because they contain the original sources.

This structure is generated directly by the parser. The parser maintains a second stack containing tree nodes that are not bound to a parent node.

As the parser passes through the token stream provided by the lexer, it generates new tree nodes and pushes them onto the stack whenever either the token is in `HIDDEN_LEXER_RULES` or the action performed is a shift action.

On a reduce action the parser will also create a new tree node. This tree node will have other tree nodes from the top of the stack as children. This behavior is very similar to how the state stack of the parser manages its state. The main difference is the tokens in the `HIDDEN_LEXER_RULES`. The reduced production may have only one symbol on the right side, but on the tree node stack there are two nodes from `HIDDEN_LEXER_RULES`. In this case the parser will pop three states from the top of the tree node stack and add them as children of the newly created node in reverse order. However, if the parser reaches the `acc` state, it is not guaranteed that the stack contains only the root node. There may be tokens from `HIDDEN_LEXER_RULES` which appeared at the beginning of the parsed string. So the parser will find the root node and prepend all other entries in the tree node stack to the children of the root node.

With this algorithm, every single token from the lexer's token list is transformed into a tree node and placed in the CST. This guarantees that the whole source code is represented in the CST and that the CST could be transformed back to the source code exactly.

3.5 AST-Generation

The AST consists of three types of nodes. `ProductionTreeNode` are nodes derived from a grammar production. `TokenTreeNode` are nodes derived from a lexer token. `StringTreeNode` are nodes that contain a string and are used to replace other tree nodes. The `StringTreeNode` is not part of the initial AST. The purpose of this node type is to replace other nodes or to be added to the AST by the user.

The AST is created from the CST. The `TokenTreeNode` which reference a token from `HIDDEN_LEXER_RULES` will marked as hidden. These nodes are stored in the AST, but are not visible to the user unless specifically requested. Next, the AST structure is modified by the modifiers defined in the grammar file. Each modifier has its own implementation and is applied bottom-up for each node.

3.5.1 list-Modifier

The `list` modifier, intended by [OJ09], is a modifier designed to flatten out self-recursive productions. This modifier can only be applied to symbols on the left side of a production.

The following example could be a production for the function parameters in common programming languages.

Listing 3: list modifier example

```
PARAMETER_LIST[list] -> PARAMETER_LIST PARAMETER |
                        PARAMETER |
                        EPSILON;
```

Without the `[list]` modifier, this would create a tree structure like this

1. `PARAMETER_LIST`
   (a) `PARAMETER_LIST`
      i. `PARAMETER`
   (b) `PARAMETER`
2. `PARAMETER`

Mostly, we do not want a structure like this. By applying the `list` modifier, the resulting structure will be:

1. `PARAMETER_LIST`
   (a) `PARAMETER`
   (b) `PARAMETER`
   (c) `PARAMETER`

This structure is much easier to manage and still contains all relevant informations about the sources.

The modifier is applied by checking if the children of the current `ProductionTreeNode` n contains a `ProductionTreeNode` m referencing the same production. If this is the case, m is replaced in the children list of n by all childrens of m.

3.5.2 alias-Modifier

The `alias` modifier, purposed by [OJ09], is a modifier that can be applied to any symbol in the right hand side of a production. This modifier, will add an alias to the tree node, which can be used to search the tree. The following might be an example for a production for addition in a programming language.

```
ADD -> NUMBER[alias=left] plus NUMBER[alias=right];
```

Without the alias modifier, the only way to differentiate between the both `NUMBER` nodes would be by the order of the children of the `ADD` tree node. The `alias` modifier allows for cleaner searches in the AST.

This modifier is applied by simply storing the alias in the tree node.

3.5.3 hidden-Modifier

The `hidden` modifier, purposed by [OJ09], is a modifier that can be applied to terminal symbols in the right hand side of a production. This modifier will hide the tree node in the AST. In the previous example of the `ADD` production, the `ADD` tree node would have a `TokenTreeNode` child, which references the `plus` lexer definition. This information is obsolute, as the production will always contain this node and the production name already contains the necessary informations. By applying the `hidden` modifier to the `plus` symbol, the corresponding tree node will still be present, but not visible unless specifically requested.

3.5.4 inline-Modifier

The `inline` modifier, purposed by [OJ09], is a modifier that can be applied to nonterminal symbols in the right hand side of a production.

By applying the modifier, the node will be replaced with all its children. This modifier should be used on nonterminals, which itself do not carry important informations, but their children do. This way all informations are maintaned, but the tree structure gets simplyfied.

3.5.5 Not implemented modifiers

[OJ09] purposed additional modifiers, those are not implemented currently. That would be the `Boolean Access` modifier, which would replace a node with a boolean value. However, as we do not generate classes for each production, this would serve little purpose for us. The same reason applies to the `superclass` modifier, which would create a hierarchy in the generated classes of the parser generator.

3.6 AST search methods

Before any modification can be applied to the AST, firstly the nodes have to be found which should be modified.

For this approach, each ast node has three methods.

1. `query: Selector -> QueryResult`
2. `queryChildren: Selector -> QueryResult`
3. `queryImmediateChildren: Selector -> QueryResult`

The `query` methods takes a selector and returns a `QueryResult`. The `QueryResult` contains all nodes in the subtree of the searched node which match the `selector`. The `QueryResult` can also contain the searched node itself.

The `queryChildren` behaves the same, but will not include the searched node itself.

The `queryImmediateChildren` will only include the immediate children of the searched node matching the `selector`.

The `QueryResult` instance also allows to perform queries on the result. This will perform the according method on all nodes in the result and create a new `QueryResult` instance containing the merged results of each node. This allows for easy chaining of selectors. Also `QueryResult` instances can be merged for further processing with the `merge` function.

3.7 Selectors

A `Selector` is a class instance that should test whether a given tree node matches the rules defined by the `Selector`. The `Selectors` can be divided into three different categories.

The first category are logical `Selectors`. They should represent logical operations. There are two `Selectors` defined by default, `AndSelector` and `OrSelector`. Each takes a list of other selectors. The `AndSelector` will only match tree nodes that match all passed `Selectors`. The `OrSelector` will match all tree nodes where at least one of the passed `Selectors` matches the node.

The second category are data `Selectors`. These selectors search for nodes based on the data they contain. They can match the name of a production, the alias of a production, the name of a lexer definition, and the value of a lexer token. The implemented selectors are

`AliasSelector`,`ProductionSelector`,`TokenSelector`,`TokenValueSelector`. All of these take a string as a parameter and check the corresponding notes for that string.

The last category are structural `Selectors`. These selectors search for nodes that have specific nodes in their children or parents.

The following example of an AST should illustrate the purpose of this category.

1. `METHOD_CALL`
    (a) `CALLED_ON`
        i. `translator`
    (b) `METHOD_NAME`
        i. `translate`

To search for all `METHOD_CALL` nodes, that call translate on a translator object, we will want these types of selectors.

An example selector for this would be

```
new AndSelector(
 new ProductionSelector("METHOD_CALL"),
 new HasImmediateChildSelector(
  new AndSelector(
   new ProductionSelector("CALLED_ON"),
   new HasImmediateChildSelector(
    new TokenValueSelector("translator")
   )
  )
 ),
 new HasImmediateChildSelector(
  new AndSelector(
   new ProductionSelector("METHOD_NAME"),
   new HasImmediateChildSelector(
    new TokenValueSelector("translate")
   )
  )
 )
)
```

By calling the `query` method on the root node of the AST, we get all the `METHOD_CALL` nodes in the AST that match those conditions.

3.8 AST Modifications

The `AbstractSyntaxTreeNode` class contains multiple methods to make modifications.

1. `replace` Replaces this node with a list of other nodes.
2. `replaceChild` Replaces a node in the childrens with a list of other nodes.
3. `remove` Removes this node
4. `removeChild` Removes a child
5. `addChild` Adds a child to the end of the children list
6. `addChildBefore` Adds a child before another node.
7. `addChildAfter` Adds a child after another node.
8. `deepClone` Clones this node and all childrens, returns a node that equals the current one but without a parent reference.

Each `AbstractSyntaxTreeNode` references their corresponding parent node. To maintain integrity, you can only add nodes that have no parent node and replace only nodes with a parent reference. To add a node to multiple other nodes, you can clone the node to be added.

3.9 Extendability

To allow for possible extensibility, we allow for several types of customization

3.9.1 Lexer

As mentioned before, lexer definitions can contain custom matchers. These matchers are custom implementations that are referenced by a name in the grammar file. These classes are written by the user and registered by calling `CustomMatcherRegistry.registerCustomMatcher`. A custom matcher must implement the `ICustomMatcher` interface. A matcher gets a `LexerContext` and returns `null` or the matched string. Using the `LexerContext` parameter, the matcher can receive the remaining input and check if the beginning of the remaining input can be tokenized by this matcher.

The following example is an implementation of a custom matcher, that would match single line comments.

Listing 4: Example of a custom matcher

```
public class ExampleCustomMatcher implements ICustomMatcher {
    @Override
    public String match(LexerContext context) {
        if(context.getNextNChars(2).equals("//")) {
            // Find the end of line. This will be the comment token
            String remaining = context.getRemainingSource();
            Pattern p = Pattern.compile(".*(\r\n|\r|\n|$)");
            Matcher matcher = p.matcher(remaining);
            if (matcher.find()) {
                return matcher.group();
            }
        }

        return null;
    }
}
```

3.9.2 AST Generation

We allow several ways to customize AST generation. First, you can implement custom modifiers for grammar symbols. You can also post-process and decorate the AST. For example, you can replace the tree nodes with your own implementation by extending `AbstractSyntaxTreeNode`.

The following example is an implementation of a tree node that is simply a string. This tree node can be used to add or replace code in the AST.

Listing 5: Implementation of StringTreeNode

```
public class StringTreeNode extends AbstractSyntaxTreeNode {
    private final String value;

    public StringTreeNode(String value) {
        this.value = value;
    }

    public String getDisplayValue() {
        return "StringTreeNode, value: " + this.value;
    }

    public String getValue() {
        return value;
```

```
    }

    protected String getSources() {
        return this.value;
    }
}
```

3.9.3 Selectors

Custom selectors can be defined by implementing `ISelector`. These can be easily defined and used with the query methods of the tree node.

The following example shows the implementation of a selector that matches a tree node based on the production used to create that node.

Listing 6: Custom selector implementation

```
public class ProductionSelector extends BaseSelector {
    private final String production;

    public ProductionSelector(String production) {
        this.production = production;
    }

    @Override
    public boolean matches(AbstractSyntaxTreeNode treeNode) {
        if(treeNode instanceof ProductionTreeNode convertedNode) {
            return convertedNode.getProduction()
                                .leftHandSymbol()
                                .name()
                                .equals(this.production);
        }
        return false;
    }
}
```

4 Grammars

4.1 MiniJava

A grammar for a java subset has been defined, known as MiniJava. This is a very limited subset of Java, but it represents common constructs found in modern programming languages. It includes basic object oriented programming, simple conditional statements, while loops and basic arithmetic operations. However, the grammar is quite limited. For example, it does not allow any other comparison operator then < and not more then one arithmetic operation without brackets. As these limitations make it hard, to create meaningfull refactorings, we have extended the syntax of MiniJava aswell to allow for other operations.

4.2 Extended MiniJava Subset

The extended MiniJava grammar includes a few major changes. The first one allows for more comparison operators. Instead of only < also `>`, `<=`, `>=`, `==` can be used. Another change is that `return` is now a normal statemet. This means, that `return` can appear anywhere in method declarations and not only anymore at the end of a method.

These changes will allow us to demonstrate two refactorings which can be done with this tool.

4.3 Automated Refactoring

Hier Beispiel/Beispiele für automatisierte Refactorings anführen. Beispielsweise Funktionsnamen von underscore zu camelCase überführen.

test

5 Use cases

In this chapter we would like to show some refactorings that have been implemented on MiniJava and some real refactoring examples, but that have not been implemented.

5.1 Yoda conditions

Yoda conditions are conditions in the form `a compop b`, where `a` is a constant and `b` is a expression. Non Yoda conditions are conditions in the form `b compop a`, where `a` is a constant and `b` is a expression.

Usually in software development, you want to follow one style and use it for the whole project. Therefore, various IDEs and other tools like lexers exist, which check the styling and if possible fix them automatically. These tools may not exist for domain specific languages, therefore a usecase of this project can be those refactorings.

To implement this refactoring, we are using the extended MiniJava syntax. At first, we are applying a selector to find all comparison operators in the AST. From the result, we can simply grab the left and right side of the node, as they are decorated with an alias. If the left side is an integer literal, we swap the left and right node. At the end, we need to swap the `compop` to the reflect the changes. < will become >=, > will become <=, <= will become >, >= will become <. == will not need to be adjusted.

With this simple code, we can adjust an entire project to not use yoda expressions. As we chose to add the whitespaces as real nodes to the AST, even though they are hidden, we do not mess up with the whitespaces. The node that corresponds to `a` does only contain the `a` constant. Any whitespaces between `a` and `compop` will be a seperate node in the AST. Therefore, by swapping a and b the formatting of the file stays the same.

5.2 Unreachable code removal

There are many cases of unreachable code, we will focus on just one case. Code that appears after a return statement in the same block, is code that will never be executed. With this refactoring, we want to remove all code, which is happening after a return statement.

This is done by creating a selector to grab all `return` statements. Once we done that, we receive the parent node of the `return`. Now we loop over the children, and after we found the `return` node, we begin to remove following nodes.

5.3 Translation system

This is an example for a refactoring, which may be required to be performed on a large scale and that is not doable by common tools.

Let a translation system be a system, which stores for a key some data. This data includes the translations for each desired language and parameters. A parameter is a variable, which gets dynamically inserted into the text. One of the parameters can be

used as a pluralization parameter.

For example, the key "apple" resolves to the english translation "{NAME} has {COUNT} apple" if the parameter `COUNT` is 1 and "{NAME} has {COUNT} apples" if `COUNT` is not 1. `\{NAME\}` is a second parameter, which does not require any pluralization.

The wanted translation is received by calling `translate("apple", {name: user, COUNT: i})`.

When switching to another translation system, which requires the pluralization parameter to be passed seperately, we are running into an issue. The desired call syntax will be:

The wanted transformed call syntax is: `translate("apple", {name: user}, {COUNT: i})`.

The old code does not include any information, which of the parameters is the pluralization parameter. We require the connection to an external source to correctly identify the wanted parameter.

There are many difficulties to perform this refactoring, and no common IDE tools can help us with this. We might be able to perform this action with regex replacements, but we would need to write or generate a seperate regex for every single translation. Even if we get each different regex, there might still be more issues, like complex definitions of the parameters and it will be not possible to parse every possibility of the call syntax, as regular expressions are not able to parse all context free grammars.

There are different options to handle this case. We could do every refactoring manually, but this will be very time consuming if there are many translations. We could also implement the adapter pattern, which would allow us to hide the new API from the old code. Or as a last option, we could implement a custom solution.

With the gp-modifiable-ast-parser we are able to parse the sources, receive an AST and identify the translation key and all parameters passed. To identify which parameter is the pluralization parameter, we can receive the definition of the translation. Afterwards, we can simply remove the pluralization parameter from the call syntax and add it as a new parameter to the function call.

5.4 Code transpilation

This project may also be used to transpile code from a source grammar to a target grammar. As an example, a transpiler was written, which transpiles applications that use the extended MiniJava grammar to applications that use the regular MiniJava grammar. An example of a transpiler which performs operations like these would be babel. This tool is used to transpile a newer version of JavaScript into an older one, which can be executed by more browser. That allows the developer to utilize new functionalities of the programming language while not breaking backwards compability.

The goal of this implementation is to show the power of a rewritable abstract syntax tree. Especially for extended return syntax, we have to apply severe modifications to the sources to make it MiniJava compatible.

5.5 Code analysis

The gp-modifiable-ast can be used to perform code analysis. Even though this was not the main intention, the requirements of the project is equal to the ones required for code analysis.

All comments and whitespaces are kept. Specific comments are often used to disable code analysis for certain parts of code. Whitespaces are required to check code formatting.

However as the gp-modifiable-ast parser is not implemented for a specific programming language, some features cannot be implemented general. For example a symbol table cannot be implemented for all programming languages in the same way.

The main benefit of using gp-modifiable-ast for code analysis would be, that some of the analysis actions could be defined ones and used for multiple programming languages.

It would still be required to pass configurations for each programming language into the analysis action.

6 Conclusion

The implemented library is capable of generating an AST, which retains all informations about the original sources. With the provided API it is possible, to traverse and modify the AST in a way to implement custom modifications to a given source. As the implemented library is not implemented to work for a specific programming language, it is not easily possible to obtain informations about certain objects. It is not easily possible to receive informations from other parts of an AST, like what type a variable is, as these informations are dependend on the source language and may not even be part of the source file.

Therefore, this library is best used for refactorings to single nodes of an AST, which contains all informations required to perform the refactoring. For more complex tasks, additional features will have to be implemented, for example a symbol table.

List of Figures

List of Tables

References

[jun] Junit 5. https://junit.org/junit5/.

[OJ09] Jeffrey L. Overbey and Ralph E. Johnson. Generating rewritable abstract syntax trees. In Dragan Gašević, Ralf Lämmel, and Eric Van Wyk, editors, *Software Language Engineering*, pages 114–133, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.